7th Workshop on Numerical Methods in Applied
Science and Engineering (NMASE 08)
Vall de Núria, 9 a 11 de enero de 2008
©LaCàN, www.lacan-upc.es

MANAGEMENT, DESIGN AND DEVELOPMENT OF A MESH GENERATION ENVIRONMENT USING OPEN SOURCE SOFTWARE

X. Roca, E.Ruiz-Gironés and J. Sarrate

Laboratori de Càlcul Numèric (LaCàN)
Departament de Matemàtica Aplicada III
Universitat Politècnica de Catalunya
Jordi Girona 1-3, E-08034 Barcelona, Spain
e-mail: {xevi.roca,eloi.ruiz,jose.sarrate}@upc.edu, web: http://www-lacan.upc.edu

Palabras clave: Mesh generation, software engineering, agile methodologies, design patterns, data structure.

Resumen. *In this paper we present an object oriented implementation of a general-purpose mesh generation environment for geometry-based simulations. The aim of this application is to unify available legacy code and new research algorithms in only one mesh generation suite. We focus in two aspects that can be of the general interest for managers, designers and developers of similar projects. On the one hand, we analyze the software engineering practices that we have followed in the management and development process. In addition, we detail and discuss the Open Source tools and libraries that we have used. On the other hand, we discuss the design and the data structure of the environment. In particular, we first summarize the topological and geometrical representation. Second, we detail our implementation of the hierarchical mesh generation structure. Third we present our design to mediate collaboration between classes. Finally, we present some of the mesh generation features to show the capabilities of the environment.*

1 INTRODUCTION

Over the past decades, significant advances have been made in the computational and rendering capabilities of graphics workstations. In parallel, an outstanding progress has been made in the development of structured and unstructured mesh generation algorithms [1, 2]. Surprisingly, mesh generation is still the biggest bottleneck in real industrial applications. Therefore, special attention has been focused on the development of graphic and interactive environment to fully automate the discretization process [3, 4, 5, 6, 7, 8].

In this work we present an object oriented implementation of a general-purpose mesh generation environment for geometry-based simulations. On the one hand, it is designed to fulfill the functional requirements of a research group on geometry-based simulations:

- *Ease of use.* The environment has to provide an intuitive interface that allows users to generate meshes with few operations.
- *Quality meshes.* The environment has to generate high quality meshes for geometry-based analysis, such as the Finite Element Method (FEM) or the Discontinous Galerkin (DG).
- *Simple and powerful geometry modeler.* The geometry modeler has to allow fast and easy modeling of middle complexity models. In addition, it has to import complex CAD designs.

On the other hand, it has to provide a convenient developing framework that verifies the following non-functional requirements:

- *Unify mesh generation code.* All mesh generation code, legacy and research, has to be integrated in the mesh generation environment.
- *Software re-use.* In order to improve productivity we have to use available libraries, adapt legacy code, and implement new classes and methods for later code re-use.
- *Scalability.* We have to promote a design and implementation that facilitates the growing of the application.
- *Maintainability.* We have to apply several software engineering practices ensuring that a small group can maintain the application.
- *Easy to code.* New developers should be productive in short-term. That is, they should able to add new features without understanding the whole of the source code.
- *Cross-platform.* The environment has to work, at least, in Windows and Linux machines.

During the management, design and development of the project, we have used several modern software engineering techniques. Therefore, in section 2 we address four aspects of the management and development process. First, we provide a detailed list of the development paradigms and techniques that we adopted to fulfill the requirement of the application. Second, we present the project management methodology that has been followed during the development of the project. Third, we summarize our selection of Open Source tools used in the analysis, design and development of the application. Fourth, we detail a list of Open Source libraries used in the environment.

In order to achieve several of the previous requirements it is of the major importance to carefully design the classes and methods of the application. Taking into account design practices [9], in section 3 we also present the design and the data structure of the mesh generation environment. Mesh data base has been widely analyzed in [10, 11, 12].

Therefore, we focus on the hierarchical mesh generation structure and how to properly manage objects collaboration. For both cases we discuss and provide class diagrams of inheritance and collaboration.

2 MANAGEMENT AND DEVELOPMENT

The management and development process of a mesh generation environment requires the use of good software engineering practices. In this section we provide useful information for managers, designers and developers of similar projects. First, we discuss programming paradigms and techniques that have been followed to develop the mesh generation environment. Second, we resume the project management process. Third, we summarize the Open Source tools that we have used to develop the environment. Finally, we detail our selection of Open Source Libraries that provide high-level features for modeling and Graphical User Interface (GUI) creation.

2.1 Development paradigms and techniques

In order to deal with software development complexity we have used several modern programming paradigms and techniques. On the one hand, they improve scalability and maintainability of the software project. On the other hand they are oriented to ensure that the development process is oriented to a clear and affordable goal.

Object oriented. We adopted the *Object Oriented programming* (OOP) paradigm to promote flexibility and scalability of the program. That is, we design and implement our application by means of objects that collaborate together. Each object is capable of receiving messages, processing data and sending messages to other objects. Several OOP languages are available. However, we use C++ [13, 14] because it is faster than other OOP languages, mature, libraries are available, and widely used in software industry.

Agile methodology. Several agile software development methodologies have appeared during last decade. They focus on development processes that are more responsive to customer needs, "agile", than traditional methods. In particular, *eXtreme Programming* [15, 16] provides traditional engineering practices and takes them to "extreme" levels. In our project we have adopted part of the eXtreme Programming practices. In particular:

- *Pair programming.* We frequently program in couples. Two programmers developing on the same machine create better quality code. The code is reviewed at the same time it is typed. Hence, the source code knowledge is shared.

- *Small steps.* Each developer is responsible of coding a particular and small feature. Developers try to implement only the required feature and not future features. We use the tickets concept of *Trac* [17] for assigning small tasks to developers.

- *Unit testing.* For each new implemented feature, a test case is added to the unit tests. Once a new feature is added all the tests in the unit are run. Unit testing ensures that after a modification is performed, the program still works and is able to run the previously developed features. We have selected *CppUnit* [18] as unit testing library.

- *Refactoring.* Developers use refactoring [19] in order to simplify and make clear the source code. Code duplication is not allowed, and each time it is found developers create adequate abstractions [9, 20]. Unit testing ensures that the program has the same functionality after refactoring.

- *Sharing the source code.* Source code is shared by all the developers in a centralized repository. The developers can verify and change any part of the source code. To properly manage code sharing we use *Subversion* [21] as a version control system.

- *Continuous integration.* After a new feature is correctly implemented, it is committed to the source code repository. That is, new features are continuously integrated in the shared source code. Subversion and distributed compilation with *distcc* [22] facilitates new features integration.

Design patterns. The development of a mesh generation environment requires the design of a complex system, with a high number of classes collaborating together. These classes collaborate through inheritance and aggregation in order to represent complex data structures such as meshes and solid models. Several issues that appear during the design process of the environment are software engineering common problems. For this reason when a design problem is identified the adequate pattern [9, 23, 24] is used to solve the problem. In addition, we take into account design patterns during code refactoring [20].

Source code guidelines and checking. Code guidelines [25, 26] help developers to write cleaner code, simplify maintenance, improve code communication, reduce coding times, and improve quality. Moreover, code guidelines are a good resource for design and programming tips [25, 27, 28, 29, 30]. In code guidelines developers find some rules on:

- *Organizational and policy issues.* Tools and techniques for writing solid code, such as: version control systems, compiler flags, code reviewing, and automatic building tools.

- *Design style.* Software engineering good design practices, such as: write simple classes, avoid premature optimization, reduce class dependencies and encapsulate data.

- *Coding style.* Coding issues, such as: how to use `#include` guards, avoid cyclic dependencies, always initialize variables, and prefer compiler and linker errors to run-time errors.

- *Implementation.* Tips and rules related to the implementation of the design concepts, such as: prefer composition to inheritance, when we have to use inheritance or templates, object construction and destruction, automatic memory management or error handling.

To verify that code guidelines are followed we use pair programming and code reviews of the new code. In addition, we automatically check part of the rules with the *Krazy* [31] tool, which is part of *English Breakfast Network* [32]. Krazy looks for some issues that should be fixed for reasons of policy, design, coding or implementation.

2.2 Project management

We adopt part of eXtreme Programming agile methodology using *Trac* [17]. Trac is an Open Source tool for web-based software project management. Moreover, it provides wiki implementation, issue tracking system and an interface to Subversion.

The flow of the development can be driven with Trac using the *Milestone* concept. Each Milestone represents a group of common *Tickets* (enhancements, tasks and defects) that define a new version (or an iteration in the eXtreme Programming context) of the program. Users of the environment can report bugs and wish lists creating new tickets. In this sense, tickets provide issue tracking functionality to Trac. The *Roadmap* shows a view of the current state of the project by means of the number of open and closed tickets per milestone. Trac allows to link tickets from: the wiki, other tickets, and commit messages of Subversion. Developers can access to the Subversion repository from a web interface. Finally, the wiki facilitates communication between developers. We use the wiki to dynamically add or change: guidelines, programming tips, comments on common programming errors, documentation and useful information for developers.

2.3 Tools

Several Open Source tools are used in order to: analyze, design, implement, manage, compile and document the source code of the environment. Below we provide our particular selection of Open Source tools.

Kubuntu [33] is a GNU/Linux distribution based on *KDE*, the K Desktop Environment [34]. KDE is build on the *Qt* library [35]. Therefore, development with Qt library is natural in this platform.

KDevelop [36] is an easy to use Integrated Development Environment (IDE) for developing KDE applications. Since KDE is based on Qt, KDevelop it is also a suitable IDE to develop with Qt. Among its features we highlight: C++ and Qt project management, debugger, profiler, and visual GUI designer.

GCC [37] is the GNU Compiler Collection. It provides compilers for several languages, in particular for C++. It is the standard compiler for development under GNU/Linux, provides a good C++ standard implementation.

distcc[22] accelerates source code builds by means of distributed compilation on several machines. Thus, it improves productivity since code, compile, test, and debug cycle time is reduced.

Subversion [21] is a version control system with similar interface and features to *CVS* [38]. However, it also provides:

- version control of directories, copies, and renames.
- revision numbers are assigned in terms of per-commit and not per-file.
- efficient operations in terms of memory and CPU time.

Moreover, several mature GUI front-ends are available for Subversion. These front-ends facilitate usual operations such as: commit, merge, blame, diff or patch. In particular, we have selected *TortoiseSVN* [39] for Windows, and *KDESvn* [40] for Kubuntu.

CMake [41], the cross-platform make, is used to automate the build process. It creates the makefiles and workspaces for a particular platform and compiler.

GDB [42], the GNU project debugger, provides standard debugging features and can be called from KDevelop.

Valgrind [43] is a complete tool for debugging and profiling Linux programs. KDevelop integrates Valgrind in the IDE. Current version provides four tools:

- *Memory error detector.* This tool allows to automatically checking for memory errors such as: uninitialized memory, bad memory access, read and write out of limits of allocated memory, and memory leaks.
- *cache (time) profiler.* It provides time cost analysis in order to improve the computational efficiency of the program.
- *Call-graph profiler.* It analyzes the call relationships between functions of an application.
- *Heap profiler.* It analyzes where, when and how much memory is allocated during the program execution.

Doxygen [44] generates, from the source code, documentation for several languages, in particular for C++. The user can configure the output and obtain documentation in: HTML, LaTeX, RTF, MS-Word, PostScript, and Unix man pages. Moreover, it can extract the source code structure and create UML inheritance and collaboration class diagrams.

2.4 Libraries

In the development process we have used the following three Open Source libraries:

- *Open CASCADE* [45] is a powerful Open Source geometry and topology kernel that provides essential features for solid modeling, CAD data exchange, and rapid application development. It is used by several Open Source mesh generation environments such as: *GMSH* [46], *SALOME* [47], and *NetGen* [48].

- *Qt* [35] is a standard cross-platform library for rapid GUI development with C++. The GUI of our environment is fully implemented with Qt. It is composed by several tool bars, dock windows and a central widget with the current 3D view.

- *GLPK* [49], the GNU Linear Programming Kit, is a C library for the resolution of large scale Linear Programming (LP) and Mixed Integer Programming (MIP) problems. This library is used to ensure edge division compatibility between adjacent faces in unstructured quadrilateral and submapping mesh generation algorithms.

3 DESIGN AND DATA STRUCTURE

Mesh generation environments require the specification, definition and implementation of a large amount of data types. Moreover, these data types have to collaborate in order to represent complex data structures as meshes, solid CAD models or mesh generation algorithms. To this end, we have adopted the programming paradigms and software engineering techniques presented in section 2. They allow us to define useful and scalable classes of objects. These objects store data and collaborate between them by means of data aggregation and inheritance. In this section we present the concepts related to these classes and an overview of their implementation.

3.1 Geometrical and topological representation

Open CASCADE library [45] provides topological and geometrical features required by the mesh generation environment:

- *Geometrical entities.* They are the geometric realizations that are used to define a domain. They are classified in four types: points, curves, surfaces, and volumes. Note that each realization may have several representations. For instance, a curve can be represented by a straight line, a B-spline or a NURB.

- *Topological entities.* They are the objects that are used to define the adjacency relationships between geometrical entities. They define the manner in which geometrical entities are composed and connected. Only one geometrical entity corresponds to each topological entity.

The data structures used to describe the geometrical and topological structure of the model follow the STEP representation [50]. Hence, one basic entity is defined for each dimension. The basic topological entities are:

- *Vertex.* Topological entity of dimension 0, whose geometrical representation is a *point.*

- *Edge.* Topological entity of dimension 1, whose geometrical representation is a *curve.* Two adjacent edges share at least one vertex.

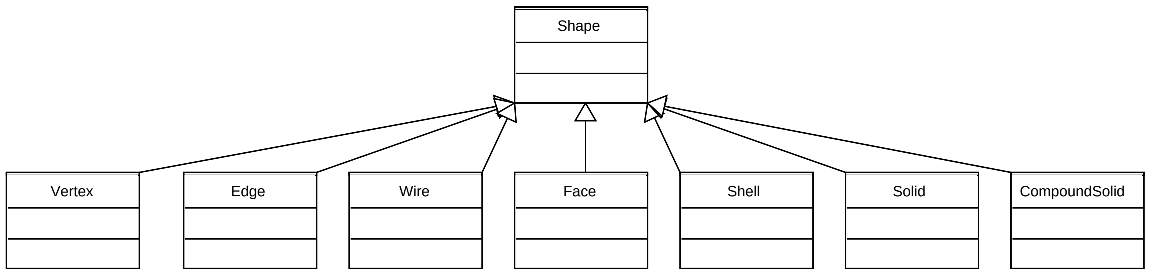

Figure 1: `Shape` class hierarchy diagram.

- *Face*. Topological entity of dimension 2, whose geometrical representation is a *surface*. Its boundary is defined by one or more loops of edges. The first loop defines the outer boundary, and the other loops define the inner holes. Two adjacent faces share at least one edge.
- *Solid*. Topological entity of dimension 3, whose geometrical representation is a *volume* with or without holes. It is defined by one or more loops of faces joined by edges. Two adjacent solids share at least one face.

Two additional topological entities are defined:

- *Wires*. Topological entity of dimension 1, determined by a closed loop of edges.
- *Shells*. Topological entity of dimension 2, determined by a closed loop of faces.
- *Compound solids*. Topological entity of dimension 3, whose determines a composition of several adjacent solids.

Object-oriented programming paradigm is used to build the interface between the topological and geometrical entities. That is, the realization of the geometrical entities is hidden (a straight line, a B-spline or a NURB for curves) of its associated topological object (an edge). Specifically, in the implementation, the `Shape` class is abstract and the rest of topological entities are specializations of this one, see figure 1. That is, all topological entities are used by means of the same function interface, determined by the abstract class `Shape`, and each particular class provides the required specialized behavior.

In order to illustrate these definitions, figure 2 shows the hierarchical representation of a surface defined by two faces, f_1 and f_2. These faces are defined by four edges, and share edge e_4. The face f_1 is defined by the set of edges $\{e_1, e_2, e_3, e_4\}$, while face f_2 is defined by the set $\{e_4, e_5, e_6, e_7\}$. Similarly, all the edges are defined by the set of vertices $\{v_1, v_2, v_3, v_4, v_5, v_6\}$. The solid lines in figure 2(b) reveal the hierarchical relationships and the shared entities. For instance, the solid lines shows that edge e_4 is shared by faces f_1 and f_2, or that vertices v_5 and v_6 are shared by three edges since three solid lines are attached to them.

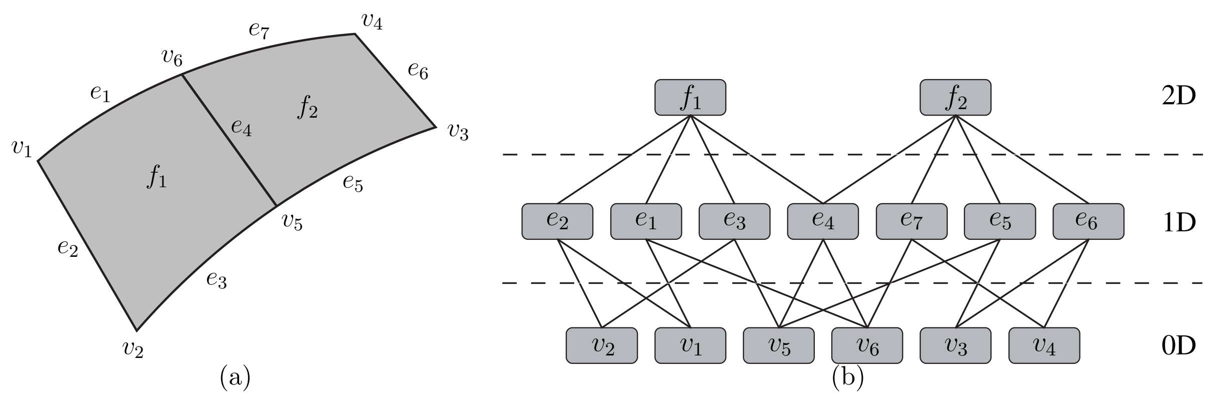

Figure 2: Topological representation of two surfaces: **(a)** A domain composed by two surfaces that share an edge; **(b)** hierarchical organization of the entities.

3.2 Mesh database

In mesh generation procedures it is required to add, remove and find mesh entities. Moreover, mesh data structure has to allow querying for adjacencies between the different mesh entities, i.e. query for the elements that surround a given node, or for the faces that share and edge. In figure 3 we present the inheritance and collaboration diagram for the `Element` and `Mesh` database classes. Thus, a mesh is composed by different types of entities together with their adjacencies. The abstract class `Element` represents the common interface for the mesh entities. Each type of mesh entity is a specialization of this abstract class. Specifically, `Node`, `Edge`, `Face` and `Cell` represent the 0D, 1D, 2D and 3D entities of a mesh, respectively. Class `Mesh` has four dynamical containers to store nodes, edges, faces and cells.

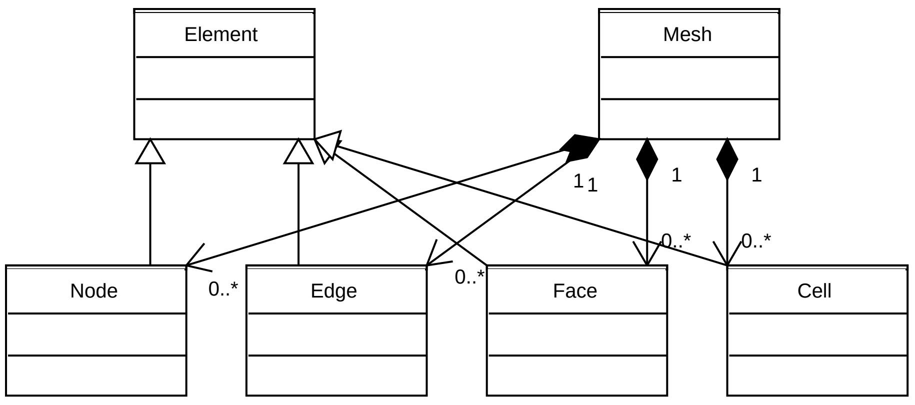

Figure 3: `Element` and `Mesh` class hierarchy and collaboration diagram.

3.3 Attributes

Mesh generation environments have to deal with some properties applied to the geometrical entities that characterize the model to simulate. These properties usually correspond to boundary conditions and material properties of the model. We denote these properties as *model attributes* and we provide an abstract and generic procedure to deal with them. The user can mark with labels the geometrical entities that compose the model in order to properly assign boundary or material conditions. Each label can be applied to several entities of the same dimension. In addition, we can apply different labels to the same entity. Moreover, the user can choose if the labels are applied over the nodes or over the elements of the entities. In the first case, we can mark with the same label the nodes of a group of vertices, edges, faces or solids. In the second case, we can mark with the same label: the edges over a group of curves; the faces over a group of surfaces; or the cells over a group of solids.

In several mesh generation environments user needs to remesh the model each time that changes or assigns new conditions. We have developed a data structure that overcomes this drawback. Specifically, mesh entities lie inside `Mesh` objects which are associated to geometrical entities. When a mesh entity asks for its attributes, this query is delegated to the container mesh that asks to corresponding geometrical entity. Not storing the value of the attributes in the mesh entities, and querying to the corresponding entity instead, allow avoiding the time consuming task of remeshing the model when a given attribute is changed.

3.4 Hierarchical mesh generation

Similar to the topological and geometrical description, the meshing algorithms are also implemented according to a hierarchical structure [51]. Therefore, the meshing algorithms are adapted to the topological and geometrical representation in a natural manner. In this sense, current implementation allows to add a new mesh generation algorithm overloading only a particular set of functions. It is important to point out that this structure is essential to ensure consistency (conformity) between meshes corresponding to adjacent entities.

The basic idea is to start by meshing entities of dimension 0 (vertices), and then proceed by meshing entities of dimension 1 (edges), dimension 2 (faces), and finally entities of dimension 3 (volumes). That is, a mesh generation algorithm is assigned to each entity of dimension d. These algorithms use as boundary mesh one or several meshes corresponding to the discretization of the boundaries entities of dimension $d-1$. These boundary meshes have been previously obtained and may be shared by other entities of dimension d. Therefore, the process always begins by meshing all the entities of lower dimension. Taking into account this property, we define four mesh generation algorithms classes: `VertexMesher`, `EdgeMesher`, `FaceMesher`, and `SolidMesher` (one for each type of topological entities). And three additional ones, `WireMesher`, `ShellMesher` and `CompoundSolidMesher` that

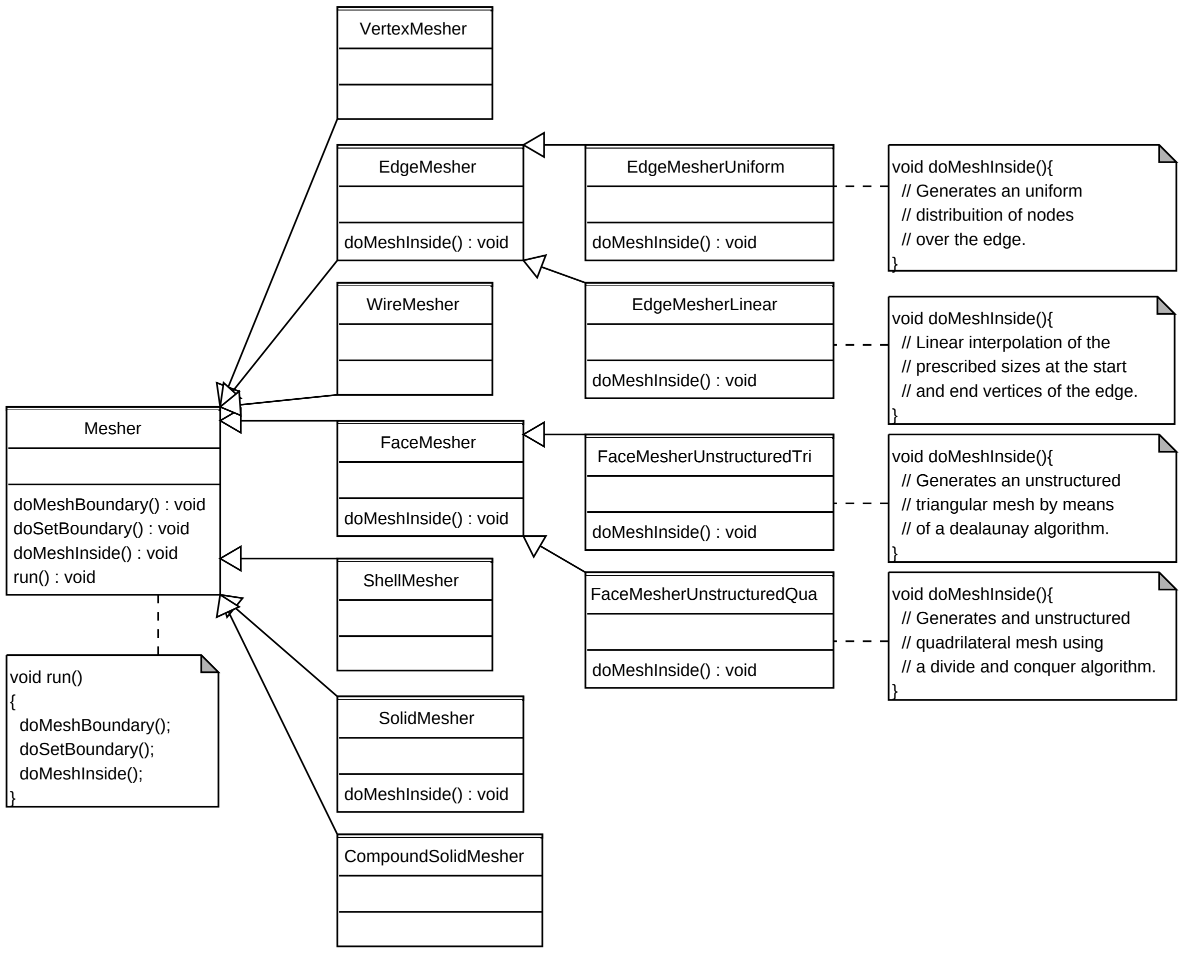

Figure 4: `Mesher` class hierarchy diagram.

allow to sew meshes over groups of edges (wires), faces (shells) and solids (compound solids).

Object oriented paradigm allows to define a natural class hierarchy for the meshers and their specializations, see figure 4. The root class is `Mesher` that provides the common interface to all meshers. A mesher is executed by means of the `run` function, that calls sequentially the not already implemented functions: `doMeshBoundary`, `doSetBoundary` and `doMeshInside`. First function, `doMeshBoundary`, calls all the meshers associated to the boundary entities of the entity to be meshed. Second function, `doSetBoundary`, adds to the mesh of the current mesher the boundary nodes obtained with `doMeshBoundary`. Third function, `doMeshInside`, calls the code of the algorithm used to mesh the inner part of the selected entity. These three functions are pure virtual, i.e. they are not implemented in the `Mesher` class and have to be defined by the particular specializations of the `Mesher`

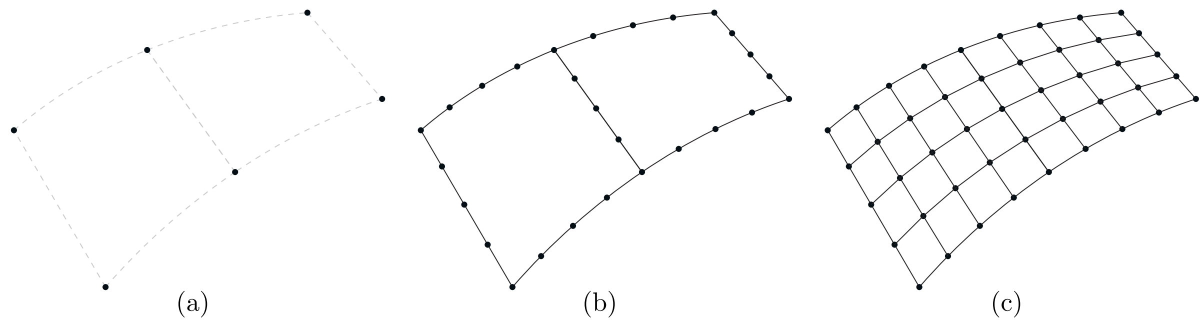

Figure 5: Hierarchical mesh generation process corresponding to the discretization of the domain presented in figure 2. **(a)** 0D mesh; **(b)** 1D mesh; and **(c)** 2D mesh.

class. The second level of classes in the hierarchy override the `doMeshBoundary` and `doSetBoundary` functions, but not `doMeshInside`. The latter is implemented by the specializations of `Mesher` that are on the third level. For instance, `EdgeMesherUniform` and `EdgeMesherLinear` override the `doMeshInside` function in order to specify a particular `EdgeMesher`. Similarly `FaceMesherUnstructuredTri` and `FaceMesherUnstructuredQua` are specializations of `FaceMesher` that allow to mesh the surface of a face with triangular or quadrilateral elements, respectively.

Figure 5 shows the hierarchical mesh generation process corresponding to the discretization of the domain presented in figure 2. In the first step, all the entities of dimension 0 (vertices) are meshed, see figure 5(a). Taking into account this discretization, in the second step the seven entities of dimension 1 (edges) are meshed, see figure 5(b). Finally, the mesh corresponding to the two entities of dimension 2 (faces) are obtained from the discretization of the edges, see figure 5(c).

3.5 Objects collaboration

In order to provide functionality to the mesh environment, the different classes have to collaborate together. This collaboration is represented by many connections between objects, which may reduce the reusability of the code. Adding more connections can lead to objects that can not work without the support of the others. Moreover, little changes in one part of the source code are propagated far away. To solve these drawbacks we introduce `Key` and `KeyManager` classes, see figure 6. The first one is devoted to mediate collaborations between the geometry and topology, attributes and meshing functions. For instance, meshers are related to one shape, and can query for their geometrical properties, such as: point coordinates, distances, curvatures or tangent vectors. Thus, assigned attributes over shapes can be obtained for a particular mesh entity. Each key has one mesher and one mesh corresponding to a particular shape. Each key has at most one background mesh, and as many attributes as it is required. The second class, `KeyManager`, is used to manage the instances of objects. Basically, it allows to add, remove and find

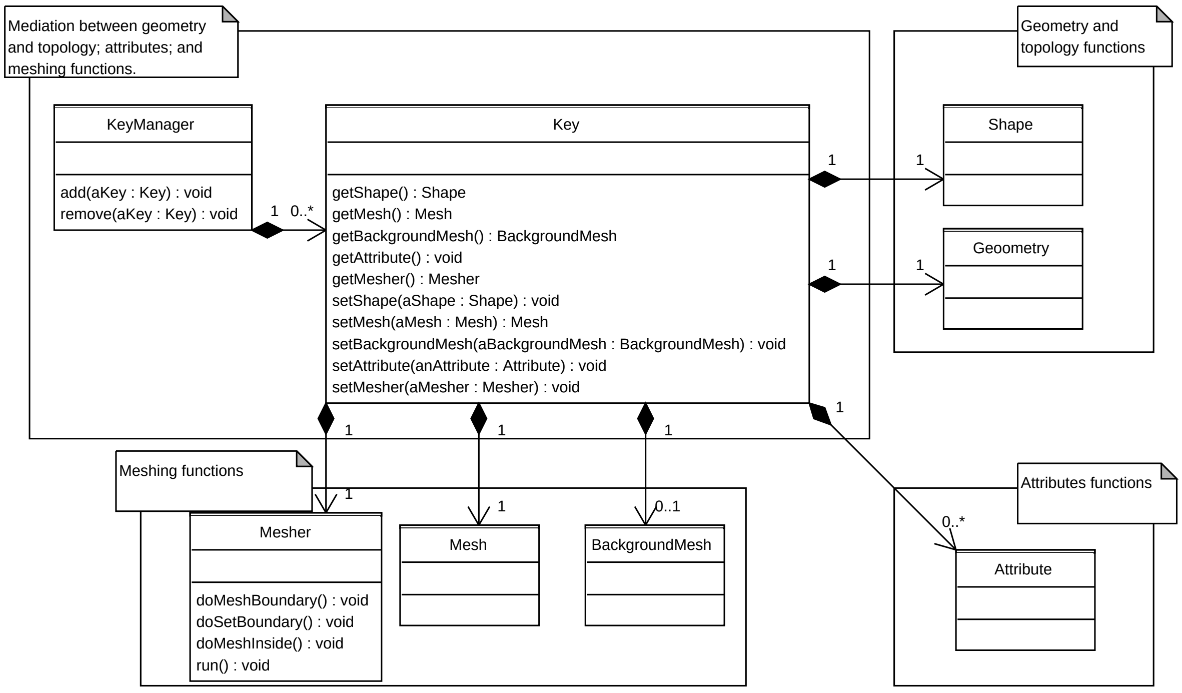
Mediation between geometry and topology; attributes; and meshing functions.
Geometry and topology functions
KeyManager
add(aKey : Key) : void
remove(aKey : Key) : void
Key
getShape() : Shape
getMesh() : Mesh
getBackgroundMesh() : BackgroundMesh
getAttribute() : void
getMesher() : Mesher
setShape(aShape : Shape) : void
setMesh(aMesh : Mesh) : Mesh
setBackgroundMesh(aBackgroundMesh : BackgroundMesh) : void
setAttribute(anAttribute : Attribute) : void
setMesher(aMesher : Mesher) : void
Shape
Geoometry
Meshing functions
Mesher
doMeshBoundary() : void
doSetBoundary() : void
doMeshInside() : void
run() : void
Mesh
BackgroundMesh
Attributes functions
Attribute
1
0..*
0..1

Figure 6: `KeyManager` and `Key` classes mediate collaboration between geometry/topology, attributes, and meshing functions.

`Key` class instances that collaborate to provide the geometry, topology, attributes and meshing features.

4 MESH GENERATION FEATURES

In this section we present some of the mesh generation features that are available at the current state of the meshing environment.

- *High-order meshes.* Several numerical formulations, such as Discontinous Galerkin (DG) and NEFEM [52], need high order discretizations. Presented environment has a high order export feature, for $p \geq 1$, that generates middle edge nodes over curves of the domain, and inner face nodes that follow curved edges of the element, see figure 7. In addition, a renumbering procedure based on the the reverse Cuthill-McKee algorithm [53] is implemented in order to renumber properly high-order meshes.

- *Quadrilateral unstructured mesh generation.* Our mesh generation environment incorporates the *Gen4U* algorithm [54, 55]. In addition, our implementation automatically assigns the parity condition prior to mesh groups of adjacent faces with quadrilateral elements. Figure 8 shows an unstructured quadrilateral mesh for a

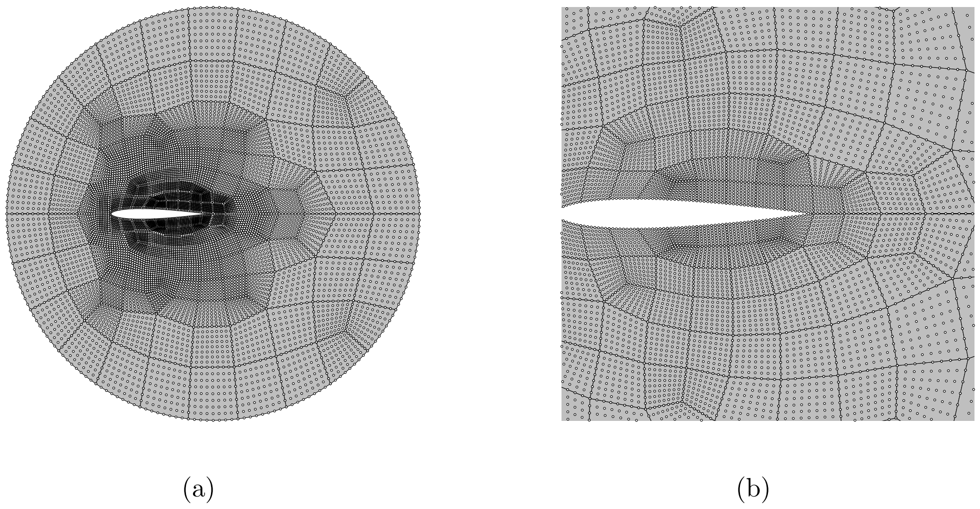
Figure 7: High order mesh, $p = 8$, for the NACA 0012: (a)11376 nodes and 175 elements, (b) detail of the mesh.

snap harness composed by 2595 nodes and 2286 quadrilaterals.

- *Sweeping.* Fully automatic unstructured hexahedral mesh generation algorithms are still not available. Therefore, special attention has been focused on existing algorithms that decompose the entire geometry into several one to one extrusion volumes. The presented mesh generation environment provides an implementation of an automatic sweep algorithm [56, 57]. Figure 9(a) shows a detail of the obtained mesh for a power chain geometry model, decomposed into several sweep volumes.

- *Submapping.* Structured meshes are preferred by several applications. Our mesh generation environment provides an implementation of the submapping algorithm [58, 59]. Figure 9(b) shows the structured hexahedral mesh automatically generated for a half part of a gear.

5 CONCLUDING REMARKS

In this paper we have presented details of the management, design and development processes of a new mesh generation tool. These processes have been guided by several modern software engineering practices. Moreover, we have used Open Source tools and libraries in order to manage and develop the application. We conclude that our selection of software engineering practices and Open Source software have allowed obtaining a first version of the environment that fulfills our initial requirements.

The presented environment is under continuous development to improve its efficiency and functionality. However, we have planned to add several new features. Currently, the CAD capabilities of our environment are based on those provided by Open CASCADE.

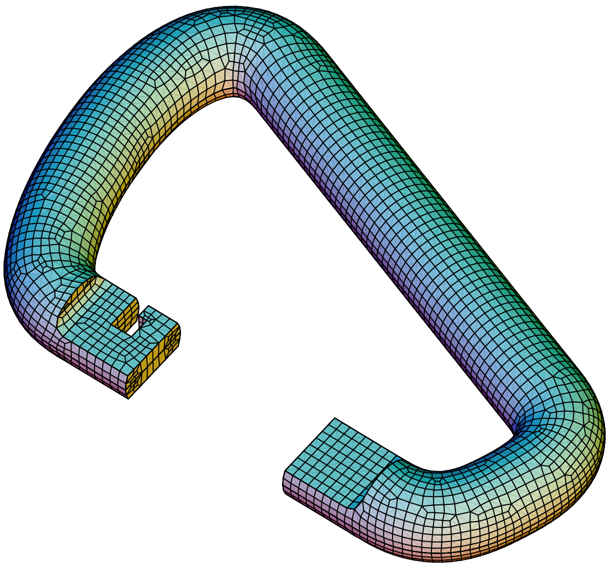

Figure 8: Unstructured quadrilateral mesh over the surface of a snap harness.

Thus, we have to supply tools for virtual geometry, feature recognition and automatic block decomposition. As a mesh generation environment, some new mesh algorithms are expected to be implemented. For instance we are implementing refinement-coarsening procedures and multi-block mesh generation algorithms. Finally, it is important to provide to developers a framework to add their own mesh generation algorithms. To this end we have decided to develop a Plug-in architecture.

REFERENCES

[1] J.F. Thompson. *Handbook of Grid Generation*. CRC Press, 1999.

[2] P.J. Frey and P.L. George. *Mesh Generation: Application to Finite Elements*. Kogan Page, 2000.

[3] M.W. Beall and M.S. Shephard. An object-oriented framework for reliable numerical simulations. *Engineering with Computers*, 15(1):61–72, 1999.

[4] M. S. Shephard. Meshing environment for geometry-based analysis. *International Journal For Numerical Methods In Engineering*, 47(1-3):169–190, January 2000.

[5] Y. Zheng, N.P. Weatherill, and E.A. Turner-Smith. An interactive geometry utility environment for multi-disciplinary computational engineering. *International Journal for Numerical Methods in Engineering*, 53:1277–1299, 2002.

[6] B. Ozell, R. Camarero, A. Garon, and F. Guibault. Analysis and visualization tools in cfd, part i: a configurable data extraction environment. *Finite Elements in Analysis and Design*, 19(4):295–307, 1995.

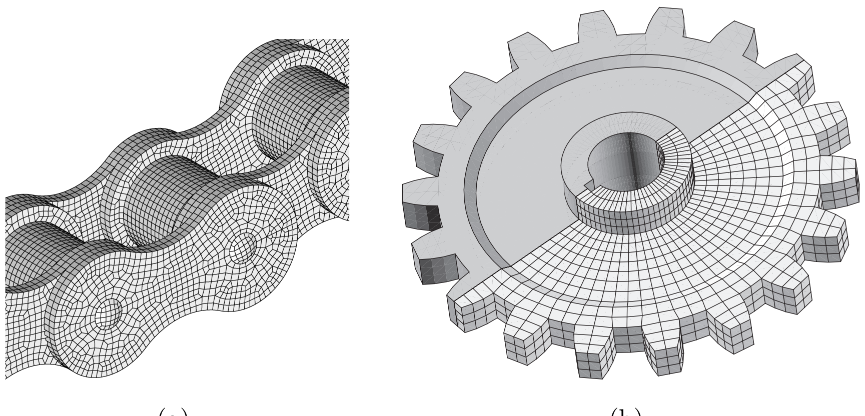
(a) (b)

Figure 9: (a) Sweeping hexahedral mesh for a power chain. (b) Submapping hexahedral mesh for the half part of a gear.

[7] S.E. Benzley, K. Merkley, T.D. Blacker, and L. Schoof. Pre-and post-processing for the finite element method. *Finite Elements in Analysis and Design*, 19(4):243–260, 1995.

[8] Cubit - geometry and mesh generation toolkit. In *http://cubit.sandia.gov*, 2007.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[10] R. V. Garimella. Mesh data structure selection for mesh generation and fea applications. *International Journal For Numerical Methods In Engineering*, 55(4):451–478, October 2002.

[11] J.F. Remacle and M.S. Shephard. An algorithm oriented mesh database. *International Journal for Numerical Methods in Engineering*, 58:349–374, 2003.

[12] T.J. Tautges. Moab-sd: integrated structured and unstructured mesh representation. *Engineering with Computers*, 20(3):286–293, 2004.

[13] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[14] A. Koenig and B.E. Moo. *Accelerated C++: practical programming by example.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[15] K. Beck. *Extreme programming explained: embrace change.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[16] J. Newkirk and R.C. Martin. *Extreme Programming in Practice.* Addison-Wesley Longman Publishing Co., Inc., 2001.

[17] The trac project - trac. In *http://trac.edgewall.org*, 2007.

[18] Cppunit - c++ port of junit. In *http://sourceforge.net/projects/cppunit*, 2007.

[19] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[20] J. Kerievsky. *Refactoring to Patterns.* Pearson Higher Education, 2004.

[21] Subversion. In *http://subversion.tigris.org*, 2007.

[22] distcc: a fast, free distributed c/c++ compiler. In *http://distcc.samba.org*, 2007.

[23] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head First Design Patterns.* O' Reilly & Associates, Inc., 2004.

[24] A. Shalloway and J. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design.* Addison-Wesley Professional, 2004.

[25] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices.* Addison-Wesley Professional, 2004.

[26] Kde developer's corner - howtos and faqs.

[27] S. Meyers. *Effective C++: 50 specific ways to improve your programs and designs.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[28] S. Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[29] H. Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions.* Pearson Higher Education, 2004.

[30] H. Sutter. *More exceptional C++: 40 new engineering puzzles, programming problems, and solutions.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[31] Krazy. In *http://techbase.kde.org/Development/Tutorials/Code_Checking*, 2007.

[32] English breakfast network. In *http://www.englishbreakfastnetwork.org*, 2007.

[33] Kubuntu - the kde desktop. In *http://www.kubuntu.org*, 2007.

[34] The k desktop environment. In *http://www.kde.org*, 2007.

[35] Qt - code less. create more. In *http://trolltech.com/products/qt*, 2007.

[36] Kdevelop - an integrated development environment. In *http://www.kdevelop.org*, 2007.

[37] Gcc, the gnu compiler collection. In *http://gcc.gnu.org*, 2007.

[38] Cvs - concurrent versions system. In *http://www.nongnu.org/cvs*, 2007.

[39] Tortoisesvn. In *http://tortoisesvn.tigris.org*, 2007.

[40] Kdesvn. In *http://www.alwins-world.de/wiki/programs/kdesvn/*, 2007.

[41] Cmake - cross platform make. In *http://www.cmake.org*, 2007.

[42] Gdb - the gnu project debugger. In *http://sourceware.org/gdb/*, 2007.

[43] Valgrind. In *http://valgrind.org*, 2007.

[44] Doxygen. In *http://www.stack.nl/ dimitri/doxygen/*, 2007.

[45] Open cascade technology, 3d modeling & numerical simulation. In *http://www.opencascade.org*, 2007.

[46] Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. In *http://www.geuz.org/gmsh/*, 2007.

[47] Salome: The open source integration platform for numerical simulation. In *http://www.salome-platform.org*, 2007.

[48] Netgen - automatic mesh generator. In *http://www.hpfem.jku.at/netgen/*, 2007.

[49] Glpk (gnu linear programming kit). In *http://www.gnu.org/software/glpk/*, 2007.

[50] Step: International standard iso 10303-42. industrial automation systems and integration - product data representation and exchange - part 42: Integrated generic resource: Geometric and topological representation. Technical report, 2000.

[51] A.N. Athanasiadis and H. Deconinck. Object-oriented three-dimensional hybrid grid generation. *International Journal for Numerical Methods in Engineering*, 58:301–318, 2003.

[52] R. Sevilla, S. Fernández-Méndez, and A. Huerta. Nurbs-enhanced finite element method. In *XIX Congreso de Ecuaciones Diferenciales y Aplicaciones, IX Congreso de la Sociedad Española de Matemática Aplicada*, 2005.

[53] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM Press New York, NY, USA, 1969.

[54] J. Sarrate and A. Huerta. Efficient unstructured quadrilateral mesh generation. *International Journal for Numerical Methods in Engineering*, 49:1327–1350, 2000.

[55] X. Roca, J. Sarrate, and A. Huerta. Generación de mallas de cuadriláteros sobre superficies paramétricas. In *Congreso de Métodos Numéricos en Ingeniería*, Granada, Spain, 2005.

[56] X. Roca, J. Sarrate, and A. Huerta. A new least squares approximation of affine mappings for sweep algorithms. In *Proccedings of the 14th International Meshing Roundtable*, 2005.

[57] X. Roca and J. Sarrate. An automatic and general least-squares projection procedure for sweep meshing. In *Proceedings of the 15th International Meshing Roundtable*, 2005.

[58] D.R. White. Automatic, quadrilateral and hexahedral meshing of pseudo-cartesian geometries using virtual subdivision. Master's thesis, Brigham Young University, 1996.

[59] E. Ruiz and J. Sarrate. Generación automática de mallas de cuadriláteros mediante programación lineal entera e interpolación transfinita. In *Congreso de Métodos Numéricos en Ingeniería*, Oporto, Portugal, 2007.

This work was partially sponsored by the *Ministerio de Ciencia y Tecnología* under grants DPI2004-03000 and CGL2004-06171-C03-01/CLI. The authors wish also to thank *Universitat Politècnica de Catalunya* and *E.T.S. d'Enginyers de Camins, Canals i Ports de Barcelona* for their support.